

Automatic Parallelization of Faust Code

Yann Orlarey Stephane Letz Dominique Fober
Grame, Centre National de Creation Musicale, France.



LAC 2009, Parma

FAUST : Functional AUdio SStream

A programming language for realtime signal processing

Goals and Principles :

FAUST : Functional AUdio SStream

A programming language for realtime signal processing

Goals and Principles :

Adequate Notation for Signal Processing

- 1 *Functional approach* : A purely functional programming language for real-time signal processing
- 2 *Strong formal basis* : A language with a well defined formal semantic

FAUST : Functional AUdio SStream

A programming language for realtime signal processing

Goals and Principles :

Adequate Notation for Signal Processing

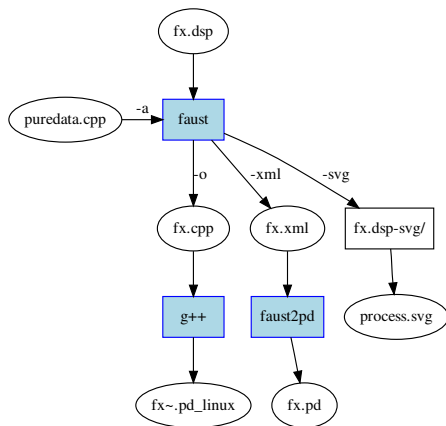
- 1 *Functional approach* : A purely functional programming language for real-time signal processing
- 2 *Strong formal basis* : A language with a well defined formal semantic

Separation between Specification and Implementation

- 1 *Efficient compiled code* : The generated C++ code should compete with hand-written code
- 2 *Easy deployment* : Multiple native implementations from a single Faust program

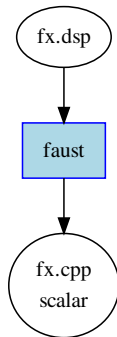
FAUST Workflow

The example of PD externals



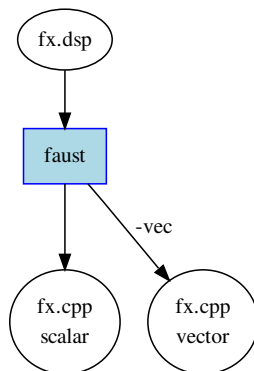
FAUST Compiler Extension

Up to FAUST 0.9.9.4 : scalar code only



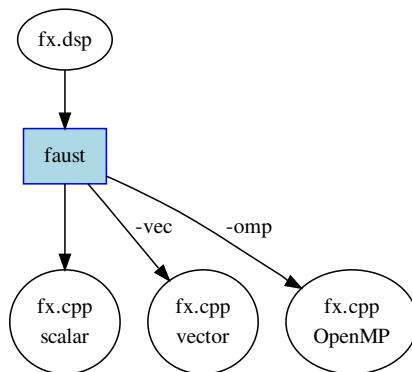
FAUST Compiler Extension

from FAUST 0.9.9.5 : vector code



FAUST Compiler Extension

from FAUST 0.9.9.5 : parallel code



FAUST Compiler Extension

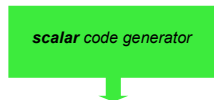
The Code Generation Stack

These code generation extensions are build on top of each other :

FAUST Compiler Extension

The Code Generation Stack

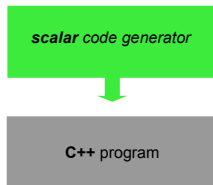
These code generation extensions are build on top of each other :



FAUST Compiler Extension

The Code Generation Stack

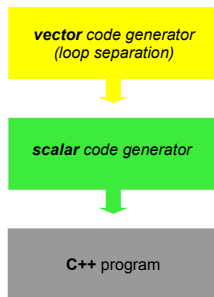
These code generation extensions are build on top of each other :



FAUST Compiler Extension

The Code Generation Stack

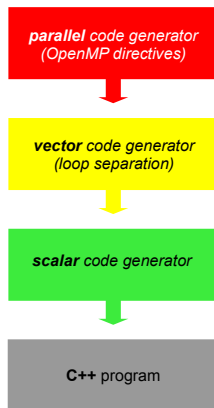
These code generation extensions are build on top of each other :



FAUST Compiler Extension

The Code Generation Stack

These code generation extensions are build on top of each other :



Scalar Compilation Scheme



A diagram showing a green rectangular box with the text *scalar code generator* inside. A green arrow points downwards from the bottom center of the box.

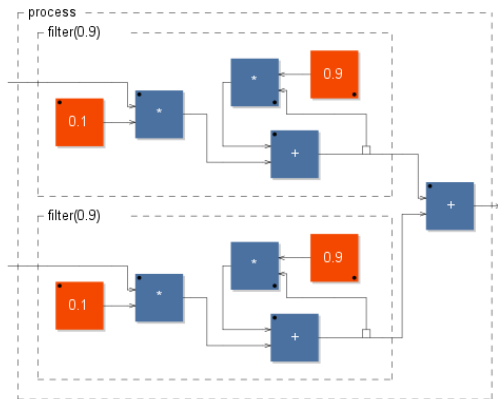
scalar code generator

The *Scalar* Compilation Scheme generates a single sample-level computation loop.

Simple Example

two 1-pole filters in parallel connected to an adder

```
filter(c) = *(1-c) : + ~ *(c);  
process = filter(0.9), filter(0.9) : +;
```



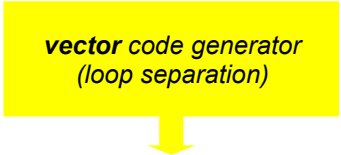
Simple Example

Scalar Code Generation

```
virtual void compute (int count, float** input,
                      float** output) {

    float* input0 = input[0];
    float* input1 = input[1];
    float* output0 = output[0];
    for (int i=0; i<count; i++) {
        fRec0[0] = (0.1f * input1[i]) + (0.9f * fRec0[1]);
        fRec1[0] = (0.1f * input0[i]) + (0.9f * fRec1[1]);
        output0[i] = (fRec1[0] + fRec0[0]);
        // post processing
        fRec1[1] = fRec1[0];
        fRec0[1] = fRec0[0];
    }
}
```

Vector Compilation Scheme



vector code generator
(loop separation)

The *Vector* Compilation Scheme simplifies the autovectorization work of the C++ compiler by splitting the sample processing loop into several simpler loops.

Simple Example

Vector Code Generation

```
...  
// SECTION : 1  
for (int i=0; i<count; i++) {  
    fRec0[i] = (0.1f * input1[i]) + (0.9f * fRec0[i-1]);  
}  
for (int i=0; i<count; i++) {  
    fRec1[i] = (0.1f * input0[i]) + (0.9f * fRec1[i-1]);  
}  
// SECTION : 2  
for (int i=0; i<count; i++) {  
    output0[i] = fRec1[i] + fRec0[i];  
}  
...
```

Parallel Compilation Scheme



parallel code generator
(OpenMP directives)

The *Parallel* Compilation Scheme analyzes the dependencies between these loops and add OpenMP pragmas to indicate those that can be computed in parallel.

OpenMP

OpenMP

OpenMP Goal

Support multi-platform shared-memory parallel programming in C/C++ and Fortran on all architectures, including Unix platforms and Windows NT platforms.

OpenMP

OpenMP Goal

Support multi-platform shared-memory parallel programming in C/C++ and Fortran on all architectures, including Unix platforms and Windows NT platforms.

Industrial Standard

Members of OpenMP ARB : AMD, Cray, Fujitsu, HP, IBM, Intel, NEC, The Portland Group, SGI, Sun Microsystems, Microsoft.

OpenMP

OpenMP Goal

Support multi-platform shared-memory parallel programming in C/C++ and Fortran on all architectures, including Unix platforms and Windows NT platforms.

Industrial Standard

Members of OpenMP ARB : AMD, Cray, Fujitsu, HP, IBM, Intel, NEC, The Portland Group, SGI, Sun Microsystems, Microsoft.

Official Web site

<http://www.openmp.org>

OpenMP

OpenMP

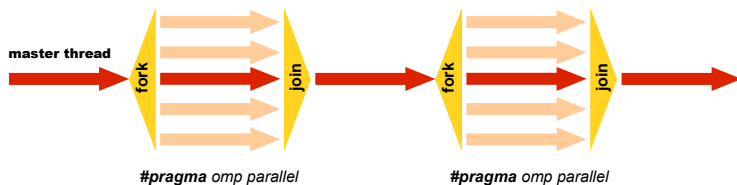
Principle

OpenMP is based on a set of compiler directives, library routines, and environment variables that influence run-time behavior in a fork-join model.

OpenMP

Principle

OpenMP is based on a set of compiler directives, library routines, and environment variables that influence run-time behavior in a fork-join model.



Simple Example

Parallel Code Generation

```
...  
// SECTION : 1  
#pragma omp sections  
{  
    #pragma omp section  
    for (int i=0; i<count; i++) {  
        fRec0[i] = (0.1f * input1[i]) + (0.9f * fRec0[i-1]);  
    }  
    #pragma omp section  
    for (int i=0; i<count; i++) {  
        fRec1[i] = (0.1f * input0[i]) + (0.9f * fRec1[i-1]);  
    }  
}  
// SECTION : 2  
#pragma omp for  
for (int i=0; i<count; i++) {  
    output0[i] = (fRec1[i] + fRec0[i]);  
}  
...
```

How do the scalar, vector and parallel code compare ?

In order to compare the new vector and parallel code with the scalar code we have run 126 tests:

How do the scalar, vector and parallel code compare ?

In order to compare the new vector and parallel code with the scalar code we have run 126 tests:

- 1 7 FAUST examples

How do the scalar, vector and parallel code compare ?

In order to compare the new vector and parallel code with the scalar code we have run 126 tests:

- 1 7 FAUST examples
- 2 3 code generations

How do the scalar, vector and parallel code compare ?

In order to compare the new vector and parallel code with the scalar code we have run 126 tests:

- 1 7 FAUST examples
- 2 3 code generations
- 3 2 compilers (gcc and icc)

How do the scalar, vector and parallel code compare ?

In order to compare the new vector and parallel code with the scalar code we have run 126 tests:

- 1 7 FAUST examples
- 2 3 code generations
- 3 2 compilers (gcc and icc)
- 4 3 machines (2, 4 and 8 cores)

What to measure ?

The tests are based on a modified Alsa/GTK architecture
`alsa-gtk-bench.cpp` that measures the duration of the
`compute()` method :

What to measure ?

The tests are based on a modified Alsa/GTK architecture `alsa-gtk-bench.cpp` that measures the duration of the `compute()` method :

TSC the duration is measured using the TSC (Time Stamp Counter) register.

What to measure ?

The tests are based on a modified Alsa/GTK architecture `alsa-gtk-bench.cpp` that measures the duration of the `compute()` method :

TSC the duration is measured using the TSC (Time Stamp Counter) register.

median A total of 128+2048 measures are made by run. The first 128 measures are considered a warm-up period and are skipped. The median value of the following 2048 measures is computed.

What to measure ?

The tests are based on a modified Alsa/GTK architecture `alsa-gtk-bench.cpp` that measures the duration of the `compute()` method :

TSC the duration is measured using the TSC (Time Stamp Counter) register.

median A total of 128+2048 measures are made by run. The first 128 measures are considered a warm-up period and are skipped. The median value of the following 2048 measures is computed.

MB/s This median value, expressed in processors cycles, is first converted in a duration, and then in number of mega-bytes produced per second (MB/s) considering the audio buffer size (in our test 2048) and the number of output channels.

Code generations

The tests are compiled with Faust 0.9.9.5b2 in three different settings :

Code generations

The tests are compiled with Faust 0.9.9.5b2 in three different settings :

```
scal faust -a alsa-gtk-bench.cpp test.dsp  
-o test.cpp
```

Code generations

The tests are compiled with Faust 0.9.9.5b2 in three different settings :

```
scal faust -a alsa-gtk-bench.cpp test.dsp  
      -o test.cpp
```

```
vec faust -a alsa-gtk-bench.cpp -vec -vs  
      3968 test.dsp -o test.cpp
```

Code generations

The tests are compiled with Faust 0.9.9.5b2 in three different settings :

```
scal faust -a alsa-gtk-bench.cpp test.dsp  
-o test.cpp
```

```
vec faust -a alsa-gtk-bench.cpp -vec -vs  
3968 test.dsp -o test.cpp
```

```
par faust -a alsa-gtk-bench.cpp -omp -vs  
3968 test.dsp -o test.cpp
```

C++ Compiler used

We have also used two different C++ compilers, GNU GCC and Intel ICC :

C++ Compiler used

We have also used two different C++ compilers, GNU GCC and Intel ICC :

gcc version 4.3.2 with options : `-O3 -march=native
-mfpmath=sse -msse -msse2 -msse3
-ffast-math -ftree-vectorize. (
-fopenmp added for OpenMP).`

C++ Compiler used

We have also used two different C++ compilers, GNU GCC and Intel ICC :

gcc version 4.3.2 with options : `-O3 -march=native -mfpmath=sse -msse -msse2 -msse3 -ffast-math -ftree-vectorize. (-fopenmp added for OpenMP).`

icc version 11.0.074 with options : `-O3 -xHost -ftz -fno-alias -fp-model fast=2. (-openmp is added for OpenMP).`

Machines used

All the tests were run on three different machines :

Machines used

All the tests were run on three different machines :

vaio a Sony Vaio SZ3VP laptop, with an Intel T7400 dual core processor at 2167 MHz, 2GB of Ram, running an Ubuntu 7.10 distribution with a 2.6.22-15-generic kernel.

Machines used

All the tests were run on three different machines :

vaio a Sony Vaio SZ3VP laptop, with an Intel T7400 dual core processor at 2167 MHz, 2GB of Ram, running an Ubuntu 7.10 distribution with a 2.6.22-15-generic kernel.

xps a Dell XPS machine with an Intel Q9300 quad core processor at 2500 MHz, 4GB of Ram, running an Ubuntu 8.10 distribution with a 2.6.27-12-generic kernel.

Machines used

All the tests were run on three different machines :

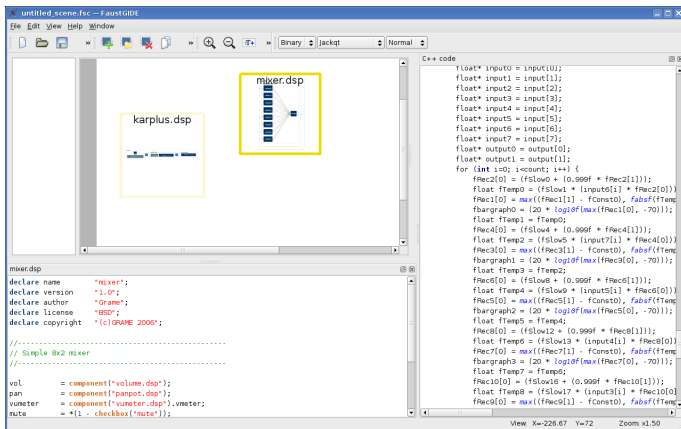
vaio a Sony Vaio SZ3VP laptop, with an Intel T7400 dual core processor at 2167 MHz, 2GB of Ram, running an Ubuntu 7.10 distribution with a 2.6.22-15-generic kernel.

xps a Dell XPS machine with an Intel Q9300 quad core processor at 2500 MHz, 4GB of Ram, running an Ubuntu 8.10 distribution with a 2.6.27-12-generic kernel.

macpro an Apple Macpro with two Intel Xeon X5365 quad core processors at 3000 MHz, 2GB of Ram, running an Ubuntu 8.10 distribution with a 2.6.27-12-generic kernel

Demo

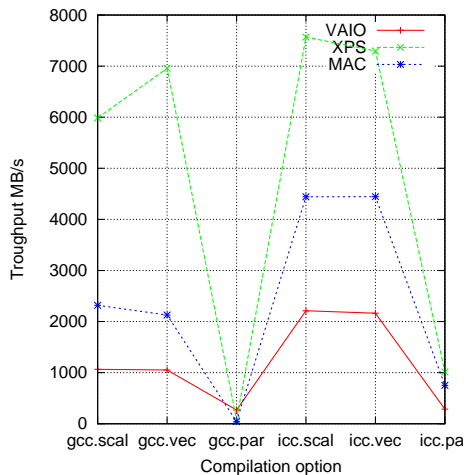
Using FAUST Graphic IDE



Copy1.dsp code

```
process = _;
```

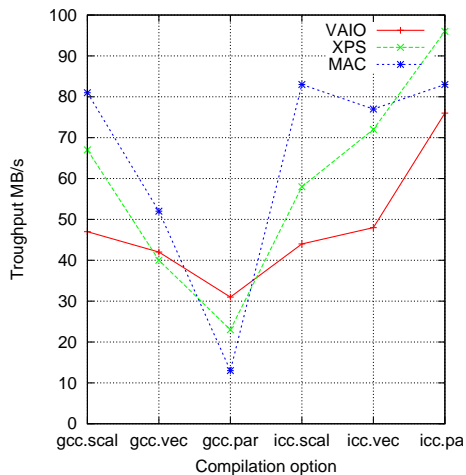
Copy1.dsp results



freeverb.dsp code

```
monoReverb(fb1, fb2, damp, spread)
= _ <:  comb(combtuningL1+spread, fb1, damp),
        comb(combtuningL2+spread, fb1, damp),
        comb(combtuningL3+spread, fb1, damp),
        comb(combtuningL4+spread, fb1, damp),
        comb(combtuningL5+spread, fb1, damp),
        comb(combtuningL6+spread, fb1, damp),
        comb(combtuningL7+spread, fb1, damp),
        comb(combtuningL8+spread, fb1, damp)
:>
    allpass (allpasstuningL1+spread, fb2)
: allpass (allpasstuningL2+spread, fb2)
: allpass (allpasstuningL3+spread, fb2)
: allpass (allpasstuningL4+spread, fb2)
;
```

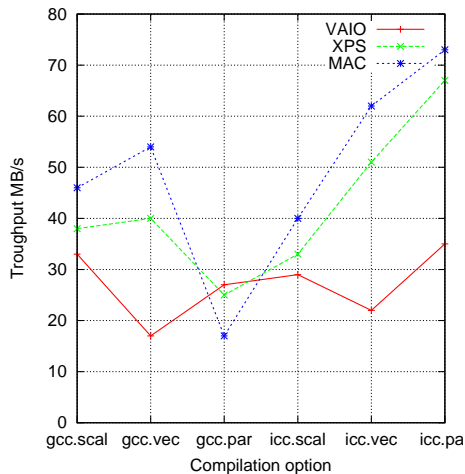
freeverb.dsp results



karplus32.dsp code

```
process =  
  vgroup("karplus32",  
    vgroup("noise_generator",  
      noise * hslider("level", 0.5, 0, 1, 0.1)  
    )  
    : vgroup("excitator",  
      *(button("play") : trigger(size))  
    )  
    <: vgroup("resonator_x32",  
      par(i,32, resonator(dur+i*detune, att)  
        * (polyphony > i)  
      )  
    )  
    :> *(output),*(output)  
  );
```

karplus32.dsp results



mixer.dsp code

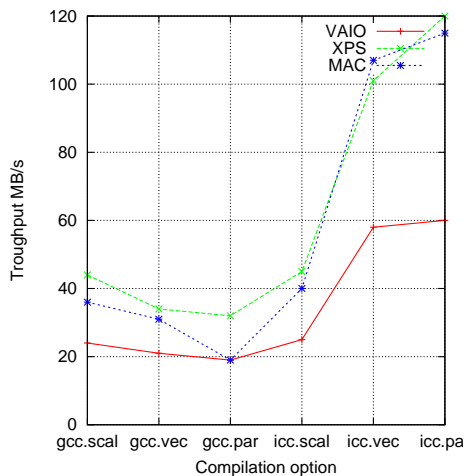
```
import("music.lib");

smooth(c) = *(1-c) : +~*(c);
vol      = *(vslider("fader", 0, -60, 4, 0.1)
             : db2linear : smooth(0.99) );
mute     = *(1 - checkbox("mute"));
vumeter(x) = attach(x, env(x) : vbargraph("",0,1))
  with{ env = abs:min(0.99):max ~ -(1.0/SR); };
pan       = _ <: *(sqrt(1-c)), *(sqrt(c))
  with{ c=(nentry("pan",0,-8,8,1)-8)/-16 : smooth(0.99);};

voice(v)  = vgroup("voice_%v",
                   mute : hgroup("", vol : vumeter) : pan );
stereo    = hgroup("stereo_out", vol, vol);

process   = hgroup("mixer", par(i,8,voice(i)) :> stereo);
```

mixer.dsp results



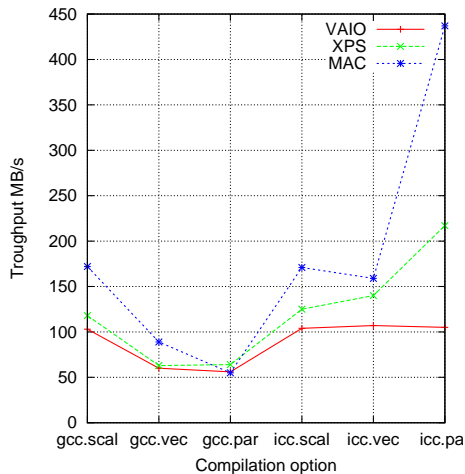
fdelay8.dsp code

```
import("filter.lib");

line(i) = vgroup("line_%"i",fdelay5(128,d):*(g))
  with{ g = vslider("gain_(dB)",-60,-60,4,0.1)
        : db2linear : smooth(0.995);
        d = nentry("delay_(samp)",10,10,128,0.1)
          : smooth(0.995);
  };

process = hgroup("", par(i, 8, line(i)) );
```

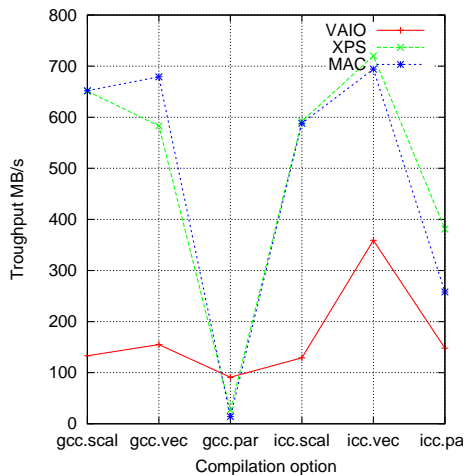
fdelay8.dsp results



rms.dsp code

```
// Square of a signal  
square(x) = x * x ;  
  
// Sliding sum of n consecutive samples  
integrate(n,x) = x - x@n : +~_ ;  
  
// Mean of n consecutive samples of a signal  
mean(n) = integrate(n) : /(n);  
  
// Root Mean Square of n consecutive samples  
RMS(n) = square : mean(n) : sqrt ;  
  
// Root Mean Square of 1000 consecutive samples  
process = RMS(1000) ;
```

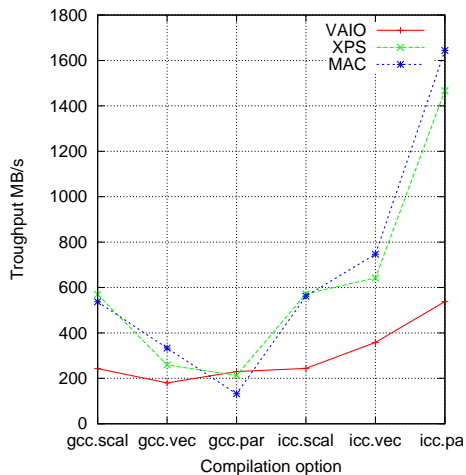
rms.dsp results



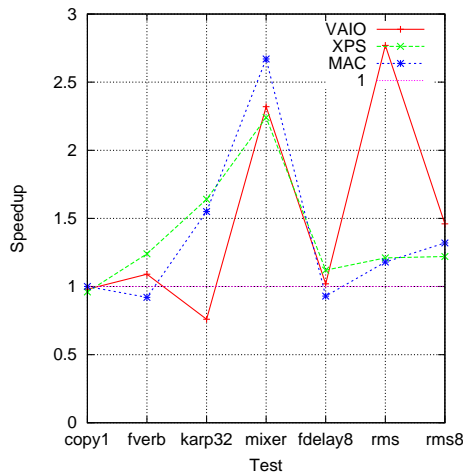
rms8.dsp code

```
process = par(i,8,component("rms.dsp")) ;
```

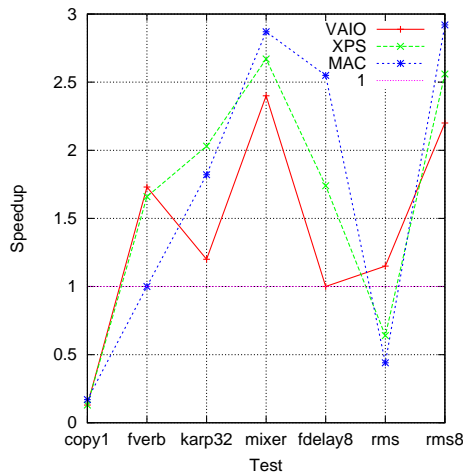
rms8.dsp results



Speedup between vector and scalar code (icc)



Speedup between parallel and scalar code (icc)



Conclusion

Automatic parallelization is the way to go :

Conclusion

Automatic parallelization is the way to go :

Discovering the parallelism of a program is :

- 1 Difficult on imperative programming languages like C/C++/Java/...
- 2 Easy on purely functional programming languages

Conclusion

Automatic parallelization is the way to go :

Discovering the parallelism of a program is :

- 1 Difficult on imperative programming languages like C/C++/Java/...
- 2 Easy on purely functional programming languages

Efficient parallelism on SMP machines is difficult

- 1 The Memory bandwidth is a strong limit and SMP doesn't scale very well
- 2 Efficient cache aware scheduling is a key factor

Conclusion

What's next ?

Improve the scheduling of the parallel tasks

Conclusion

What's next ?

Improve the scheduling of the parallel tasks

- 1 OpenMP 3.0 tasks

Conclusion

What's next ?

Improve the scheduling of the parallel tasks

- 1 OpenMP 3.0 tasks
- 2 Intel **TBB** (ThreadingBuildingBlocks):
<http://www.threadingbuildingblocks.org>

Conclusion

What's next ?

Improve the scheduling of the parallel tasks

- 1 OpenMP 3.0 tasks
- 2 Intel **TBB** (ThreadingBuildingBlocks):
<http://www.threadingbuildingblocks.org>
- 3 Cilk Arts **Cilk++**: <http://www.cilk.com>

Conclusion

What's next ?

Improve the scheduling of the parallel tasks

- 1 OpenMP 3.0 tasks
- 2 Intel **TBB** (ThreadingBuildingBlocks):
<http://www.threadingbuildingblocks.org>
- 3 Cilk Arts **Cilk++**: <http://www.cilk.com>
- 4 Develop a new scheduling algorithm (derived from work stealing schedulers)